Working with Different Types of Data in R

*Wait, there are different types of data?*

R is a flexible language that allows you to work with many different *forms* of data. This includes numeric, character, categorical, dates, and logical. Technically, R classifies all the different types of data into five classes:

* integer
* numeric
* character
* complex
* logical

Modern day analysis typically deals with every class so its important to gain fluency in dealing with these data forms. This section covers the fundamentals of handling the different data classes. First I cover the basics of dealing with [numbers](#numbers) so you understand the different classes of numbers, how to generate number sequences, compare numeric values, and round. I then provide an introduction to working with [characters](#characters) to get you comfortable with character string manipulation and set operations. This prepares you to then learn about [regular expressions](#regex) which deals with search patterns for character classes. Next I introduce [factors](#factors), also referred to as categorical variables, and how to create, convert, order, and re-level this data class. Lastly, I cover how to manage [dates](#dates) as this can be a persnickety type of variable when performing data analysis. Throughout several of these chapters you'll also gain an understanding of the TRUE/FALSE logical variables.

Together, this will give you a solid foundation for dealing with the basic data classes in R so that when you start to learn how to manage the different data structures, which combines these data classes into multiple dimensions, you will have a strong base from which to start.

# Dealing with Numbers

In this chapter you will learn the basics of working with numbers in R. This includes understanding how to manage the [numeric type (integer vs. double)](#integer_vs_double), the different ways of generating [non-random](#generating_sequence_numbers) and [random](#generating_random_numbers) numbers, how to [set seed values](#setting_seed) for reproducible random number generation, and the different ways to [compare](#compare_numeric_values) and [round](#round_numbers) numeric values.

## Integer vs. Double

The two most common numeric classes used in R are integer and double (for double precision floating point numbers). R automatically converts between these two classes when needed for mathematical purposes. As a result, it's feasible to use R and perform analyses for years without specifying these differences. To check whether a pre-existing vector is made up of integer or double values you can use typeof(x) which will tell you if the vector is a double, integer, logical, or character type.

### Creating Integer and Double Vectors

By default, when you create a numeric vector using the c() function it will produce a vector of double precision numeric values. To create a vector of integers using c() you must specify explicity by placing an L directly after each number.

# create a string of double-precision values  
dbl\_var <- c(1, 2.5, 4.5)   
dbl\_var

## [1] 1.0 2.5 4.5

# placing an L after the values creates a string of integers  
int\_var <- c(1L, 6L, 10L)  
int\_var

## [1] 1 6 10

### Converting Between Integer and Double Values

By default, if you read in data that has no decimal points or you [create numeric values](#generating_sequence_numbers) using the x <- 1:10 method the numeric values will be coded as integer. If you want to change a double to an integer or vice versa you can specify one of the following:

# converts integers to double-precision values  
as.double(int\_var)

## [1] 1 6 10

# identical to as.double()  
as.numeric(int\_var)

## [1] 1 6 10

# converts doubles to integers  
as.integer(dbl\_var)

## [1] 1 2 4

## Generating sequence of non-random numbers

There are a few R operators and functions that are especially useful for creating vectors of non-random numbers. These functions provide multiple ways for generating sequences of numbers.

### Specifing Numbers within a Sequence

To explicitly specify numbers in a sequence you can use the colon : operator to specify all integers between two specified numbers or the combine c() function to explicity specify all numbers in the sequence.

# create a vector of integers between 1 and 10  
1:10

## [1] 1 2 3 4 5 6 7 8 9 10

# create a vector consisting of 1, 5, and 10  
c(1, 5, 10)

## [1] 1 5 10

# save the vector of integers between 1 and 10 as object x  
x <- 1:10   
x

## [1] 1 2 3 4 5 6 7 8 9 10

### Generating Regular Sequences

A generalization of : is the seq() function, which generates a sequence of numbers with a specified arithmetic progression.

# generate a sequence of numbers from 1 to 21 by increments of 2  
seq(from = 1, to = 21, by = 2)

## [1] 1 3 5 7 9 11 13 15 17 19 21

# generate a sequence of numbers from 1 to 21 that has 15 equal   
# incremented numbers  
seq(0, 21, length.out = 15)

## [1] 0.0 1.5 3.0 4.5 6.0 7.5 9.0 10.5 12.0 13.5 15.0 16.5 18.0 19.5  
## [15] 21.0

The rep() function allows us to conveniently repeat specified constants into long vectors. This function allows for collated and non-collated repetitions.

# replicates the values in x a specified number of times  
rep(1:4, times = 2)

## [1] 1 2 3 4 1 2 3 4

# replicates the values in x in a collated fashion  
rep(1:4, each = 2)

## [1] 1 1 2 2 3 3 4 4

## Generating sequence of random numbers

Simulation is a common practice in data analysis. Sometimes your analysis requires the implementation of a statistical procedure that requires random number generation or sampling (i.e. Monte Carlo simulation, bootstrap sampling, etc). R comes with a set of pseudo-random number generators that allow you to simulate the most common probability distributions such as Uniform, Normal, Binomial, Poisson, Exponential and Gamma.

### Uniform numbers

To generate random numbers from a uniform distribution you can use the runif() function. Alternatively, you can use sample() to take a random sample using with or without replacements.

# generate n random numbers between the default values of 0 and 1  
runif(n)   
  
# generate n random numbers between 0 and 25  
runif(n, min = 0, max = 25)   
  
# generate n random numbers between 0 and 25 (with replacement)  
sample(0:25, n, replace = TRUE)   
  
# generate n random numbers between 0 and 25 (without replacement)  
sample(0:25, n, replace = FALSE)

For example, to generate 25 random numbers between the values 0 and 10:

runif(25, min = 0, max = 10)

## [1] 6.11473003 9.72918761 0.04977565 0.98291110 8.53146606 1.17408103  
## [7] 1.09907810 5.83266343 8.04336903 1.70783108 3.13275943 1.28380380  
## [13] 8.67087873 8.02653947 7.23398025 4.62386458 3.03617622 6.10895175  
## [19] 6.39970018 9.02183043 3.24990736 4.64181107 5.35496769 9.97374324  
## [25] 3.30954880

For each non-uniform probability distribution there are four primary functions available to generate random numbers, density (aka probability mass function), cumulative density, and quantiles. The prefixes for these functions are:

* r: random number generation
* d: density or probability mass function
* p: cumulative distribution
* q: quantiles

### Normal Distribution Numbers

The normal (or Gaussian) distribution is the most common and well know distribution. Within R, the normal distribution functions are written as norm().

# generate n random numbers from a normal distribution with given   
# mean and standard deviation  
rnorm(n, mean = 0, sd = 1)   
  
# generate CDF probabilities for value(s) in vector q   
pnorm(q, mean = 0, sd = 1)   
  
# generate quantile for probabilities in vector p  
qnorm(p, mean = 0, sd = 1)   
  
# generate density function probabilites for value(s) in vector x  
dnorm(x, mean = 0, sd = 1)

For example, to generate 25 random numbers from a normal distribution with mean = 100 and standard deviation = 15:

x <- rnorm(25, mean = 100, sd = 15)   
x

## [1] 97.43216 98.98658 96.43514 73.77727 100.51316 103.11050 111.36823  
## [8] 102.09288 101.16769 114.54549 99.28044 97.51866 110.57522 87.85074  
## [15] 86.67675 108.95660 88.45750 106.28923 114.22225 80.17450 110.39667  
## [22] 96.87112 112.30709 110.54963 93.24365

summary(x)

## Min. 1st Qu. Median Mean 3rd Qu. Max.   
## 73.78 96.44 100.50 100.10 110.40 114.50

You can also pass a vector of values. For instance, say you want to know the CDF probabilities for each value in the vector x created above:

pnorm(x, mean = 100, sd = 15)

## [1] 0.43203732 0.47306731 0.40607337 0.04021628 0.51364538 0.58213815  
## [7] 0.77573919 0.55548261 0.53102479 0.83390182 0.48086992 0.43430567  
## [13] 0.75959941 0.20898424 0.18721209 0.72478191 0.22079836 0.66249503  
## [19] 0.82847339 0.09313407 0.75588023 0.41738339 0.79402667 0.75906822  
## [25] 0.32620260

### Binomial Distribution Numbers

This is conventionally interpreted as the number of successes in size = x trials and with prob = p probability of success:

# generate a vector of length n displaying the number of successes   
# from a trial size = 100 with a probabilty of success = 0.5  
rbinom(n, size = 100, prob = 0.5)   
  
# generate CDF probabilities for value(s) in vector q  
pbinom(q, size = 100, prob = 0.5)   
  
# generate quantile for probabilities in vector p  
qbinom(p, size = 100, prob = 0.5)   
  
# generate density function probabilites for value(s) in vector x  
dbinom(x, size = 100, prob = 0.5)

### Poisson Distribution Numbers

The Poisson distribution is a discrete probability distribution that expresses the probability of a given number of events occuring in a fixed interval of time and/or space if these events occur with a known average rate and independently of the time since the last event.

# generate a vector of length n displaying the random number of   
# events occuring when lambda (mean rate) equals 4.  
rpois(n, lambda = 4)   
  
# generate CDF probabilities for value(s) in vector q when lambda   
# (mean rate) equals 4.  
ppois(q, lambda = 4)   
  
# generate quantile for probabilities in vector p when lambda   
# (mean rate) equals 4.  
qpois(p, lambda = 4)   
  
# generate density function probabilites for value(s) in vector x   
# when lambda (mean rate) equals 4.  
dpois(x, lambda = 4)

### Exponential Distribution Numbers

The Exponential probability distribution describes the time between events in a Poisson process.

# generate a vector of length n with rate = 1  
rexp(n, rate = 1)   
  
# generate CDF probabilities for value(s) in vector q when rate = 4.  
pexp(q, rate = 1)   
  
# generate quantile for probabilities in vector p when rate = 4.  
qexp(p, rate = 1)   
  
# generate density function probabilites for value(s) in vector x   
# when rate = 4.  
dexp(x, rate = 1)

### Gamma Distribution Numbers

The Gamma probability distribution is related to the Beta distribution and arises naturally in processes for which the waiting times between Poisson distributed events are relevant.

# generate a vector of length n with shape parameter = 1  
rgamma(n, shape = 1)   
  
# generate CDF probabilities for value(s) in vector q when shape   
# parameter = 1.  
pgamma(q, shape = 1)   
  
# generate quantile for probabilities in vector p when shape   
# parameter = 1.  
qgamma(p, shape = 1)   
  
# generate density function probabilites for value(s) in vector x   
# when shape   
# parameter = 1.  
dgamma(x, shape = 1)

## Setting the seed for reproducible random numbers

If you want to generate a sequence of random numbers and then be able to reproduce that same sequence of random numbers later you can set the random number seed generator with set.seed(). This is a critical aspect of [reproducible research](https://en.wikipedia.org/wiki/Reproducibility).

For example, we can reproduce a random generation of 10 values from a normal distribution:

set.seed(197)  
rnorm(n = 10, mean = 0, sd = 1)

## [1] 0.6091700 -1.4391423 2.0703326 0.7089004 0.6455311 0.7290563  
## [7] -0.4658103 0.5971364 -0.5135480 -0.1866703

set.seed(197)  
rnorm(n = 10, mean = 0, sd = 1)

## [1] 0.6091700 -1.4391423 2.0703326 0.7089004 0.6455311 0.7290563  
## [7] -0.4658103 0.5971364 -0.5135480 -0.1866703

## Comparing numeric values

There are multiple ways to compare numeric values and vectors. This includes [logical operators](#numeric_comparison) along with testing for [exact equality](#numeric_exact) and also [near equality](#numeric_near).

### Comparison Operators

The normal binary operators allow you to compare numeric values and provides the answer in logical form:

x < y # is x less than y  
x > y # is x greater than y  
x <= y # is x less than or equal to y  
x >= y # is x greater than or equal to y  
x == y # is x equal to y  
x != y # is x not equal to y

These operations can be used for single number comparison:

x <- 9  
y <- 10  
  
x == y

## [1] FALSE

and also for comparison of numbers within vectors:

x <- c(1, 4, 9, 12)  
y <- c(4, 4, 9, 13)  
  
x == y

## [1] FALSE TRUE TRUE FALSE

Note that logical values TRUE and FALSE equate to 1 and 0 respectively. So if you want to identify the number of equal values in two vectors you can wrap the operation in the sum() function:

# How many pairwise equal values are in vectors x and y  
sum(x == y)

## [1] 2

If you need to identify the location of pairwise equalities in two vectors you can wrap the operation in the which() function:

# Where are the pairwise equal values located in vectors x and y  
which(x == y)

## [1] 2 3

### Exact Equality

To test if two objects are exactly equal:

x <- c(4, 4, 9, 12)  
y <- c(4, 4, 9, 13)  
  
identical(x, y)

## [1] FALSE

x <- c(4, 4, 9, 12)  
y <- c(4, 4, 9, 12)  
  
identical(x, y)

## [1] TRUE

### Floating Point Comparison

Sometimes you wish to test for 'near equality'. The all.equal() function allows you to test for equality with a difference tolerance of 1.5e-8.

x <- c(4.00000005, 4.00000008)  
y <- c(4.00000002, 4.00000006)  
  
all.equal(x, y)

## [1] TRUE

If the difference is greater than the tolerance level the function will return the mean relative difference:

x <- c(4.005, 4.0008)  
y <- c(4.002, 4.0006)  
  
all.equal(x, y)

## [1] "Mean relative difference: 0.0003997102"

## Rounding numbers

There are many ways of rounding to the nearest integer, up, down, or toward a specified decimal place. The following illustrates the common ways to round.

x <- c(1, 1.35, 1.7, 2.05, 2.4, 2.75, 3.1, 3.45, 3.8, 4.15,   
 4.5, 4.85, 5.2, 5.55, 5.9)  
  
# Round to the nearest integer  
round(x)

## [1] 1 1 2 2 2 3 3 3 4 4 4 5 5 6 6

# Round up  
ceiling(x)

## [1] 1 2 2 3 3 3 4 4 4 5 5 5 6 6 6

# Round down  
floor(x)

## [1] 1 1 1 2 2 2 3 3 3 4 4 4 5 5 5

# Round to a specified decimal  
round(x, digits = 1)

## [1] 1.0 1.4 1.7 2.0 2.4 2.8 3.1 3.5 3.8 4.2 4.5 4.8 5.2 5.5 5.9

# Dealing with Character Strings

Dealing with character strings is often under-emphasized in data analysis training. The focus typically remains on numeric values; however, the growth in data collection is also resulting in greater bits of information embedded in character strings. Consequently, handling, cleaning and processing character strings is becoming a prerequisite in daily data analysis. This chapter is meant to give you the foundation of working with characters by covering some [basics](#character_basics) followed by learning how to [manipulate strings](#string_manipulation) using base R functions along with using the simplified stringr package.

## Character string basics

In this section you'll learn the basics of creating, converting and printing character strings followed by how to assess the number of elements and characters in a string.

### Creating Strings

The most basic way to create strings is to use quotation marks and assign a string to an object similar to creating number sequences.

a <- "learning to create" # create string a  
b <- "character strings" # create string b

The paste() function provides a versatile means for creating and building strings. It takes one or more R objects, converts them to "character", and then it concatenates (pastes) them to form one or several character strings.

# paste together string a & b  
paste(a, b)

## [1] "learning to create character strings"

# paste character and number strings (converts numbers to   
# character class)  
paste("The life of", pi)

## [1] "The life of 3.14159265358979"

# paste multiple strings  
paste("I", "love", "R")

## [1] "I love R"

# paste multiple strings with a separating character  
paste("I", "love", "R", sep = "-")

## [1] "I-love-R"

# use paste0() to paste without spaces btwn characters  
paste0("I", "love", "R")

## [1] "IloveR"

# paste objects with different lengths  
paste("R", 1:5, sep = " v1.")

## [1] "R v1.1" "R v1.2" "R v1.3" "R v1.4" "R v1.5"

### Converting to Strings

Test if strings are characters with is.character() and convert strings to character with as.character() or with toString().

a <- "The life of"   
b <- pi  
  
is.character(a)

## [1] TRUE

is.character(b)

## [1] FALSE

c <- as.character(b)  
is.character(c)

## [1] TRUE

toString(c("Aug", 24, 1980))

## [1] "Aug, 24, 1980"

### Printing Strings

The common printing methods include:

* print(): generic printing
* noquote(): print with no quotes
* cat(): concatenate and print with no quotes
* sprintf(): a wrapper for the C function sprintf, that returns a character vector containing a formatted combination of text and variable values

The primary printing function in R is print()

x <- "learning to print strings"   
  
# basic printing  
print(x)

## [1] "learning to print strings"

# print without quotes  
print(x, quote = FALSE)

## [1] learning to print strings

An alternative to printing a string without quotes is to use noquote()

noquote(x)

## [1] learning to print strings

Another very useful function is cat() which allows us to concatenate objects and print them either on screen or to a file. The output result is very similar to noquote(); however, cat() does not print the numeric line indicator. As a result, cat() can be useful for printing nicely formated responses to users.

# basic printing (similar to noquote)  
cat(x)

## learning to print strings

# combining character strings  
cat(x, "in R")

## learning to print strings in R

# basic printing of alphabet  
cat(letters)

## a b c d e f g h i j k l m n o p q r s t u v w x y z

# specify a seperator between the combined characters  
cat(letters, sep = "-")

## a-b-c-d-e-f-g-h-i-j-k-l-m-n-o-p-q-r-s-t-u-v-w-x-y-z

# collapse the space between the combine characters  
cat(letters, sep = "")

## abcdefghijklmnopqrstuvwxyz

You can also format the line width for printing long strings using the fill argument:

x <- "Today I am learning how to print strings."  
y <- "Tomorrow I plan to learn about textual analysis."  
z <- "The day after I will take a break and drink a beer."  
  
cat(x, y, z, fill = 0)

## Warning in cat(x, y, z, fill = 0): non-positive 'fill' argument will be  
## ignored

## Today I am learning how to print strings. Tomorrow I plan to learn about textual analysis. The day after I will take a break and drink a beer.

cat(x, y, z, fill = 5)

## Today I am learning how to print strings.   
## Tomorrow I plan to learn about textual analysis.   
## The day after I will take a break and drink a beer.

sprintf() is a useful printing function for precise control of the output. It is a wrapper for the C function sprintf and returns a character vector containing a formatted combination of text and variable values.

To substitute in a string or string variable, use %s:

x <- "print strings"  
  
# substitute a single string/variable  
sprintf("Learning to %s in R", x)

## [1] "Learning to print strings in R"

# substitute multiple strings/variables  
y <- "in R"  
sprintf("Learning to %s %s", x, y)

## [1] "Learning to print strings in R"

For integers, use %d or a variant:

version <- 3  
  
# substitute integer  
sprintf("This is R version:%d", version)

## [1] "This is R version:3"

# print with leading spaces  
sprintf("This is R version:%4d", version)

## [1] "This is R version: 3"

# can also lead with zeros  
sprintf("This is R version:%04d", version)

## [1] "This is R version:0003"

For floating-point numbers, use %f for standard notation, and %e or %E for exponential notation:

sprintf("%f", pi) # '%f' indicates 'fixed point' decimal notation

## [1] "3.141593"

sprintf("%.3f", pi) # decimal notation with 3 decimal digits

## [1] "3.142"

sprintf("%1.0f", pi) # 1 integer and 0 decimal digits

## [1] "3"

sprintf("%5.1f", pi) # decimal notation with 5 total decimal digits and

## [1] " 3.1"

# only 1 to the right of the decimal point  
  
sprintf("%05.1f", pi) # same as above but fill empty digits with zeros

## [1] "003.1"

sprintf("%+f", pi) # print with sign (positive)

## [1] "+3.141593"

sprintf("% f", pi) # prefix a space

## [1] " 3.141593"

sprintf("%e", pi) # exponential decimal notation 'e'

## [1] "3.141593e+00"

sprintf("%E", pi) # exponential decimal notation 'E'

## [1] "3.141593E+00"

### Counting string elements and characters

To count the number of elements in a string use length():

length("How many elements are in this string?")

## [1] 1

length(c("How", "many", "elements", "are", "in", "this", "string?"))

## [1] 7

To count the number of characters in a string use nchar():

nchar("How many characters are in this string?")

## [1] 39

nchar(c("How", "many", "characters", "are", "in", "this", "string?"))

## [1] 3 4 10 3 2 4 7

## String manipulation with base R

Basic string manipulation typically inludes case conversion, simple character and substring replacement, adding/removing whitespace, and performing set operations to compare similarities and differences between two character vectors. These operations can all be performed with base R functions; however, some operations (or at least their syntax) are simplified with the stringr package which we will discuss in the next section. This section illustrates the base R string manipulation capabilities.

### Case conversion

To convert all upper case characters to lower case use tolower():

x <- "Learning To MANIPULATE strinGS in R"  
  
tolower(x)

## [1] "learning to manipulate strings in r"

To convert all lower case characters to upper case use toupper():

toupper(x)

## [1] "LEARNING TO MANIPULATE STRINGS IN R"

### Simple Character Replacement

To replace a character (or multiple characters) in a string you can use chartr():

# replace 'A' with 'a'  
x <- "This is A string."  
chartr(old = "A", new = "a", x)

## [1] "This is a string."

# multiple character replacements  
# replace any 'd' with 't' and any 'z' with 'a'  
y <- "Tomorrow I plzn do lezrn zbout dexduzl znzlysis."  
chartr(old = "dz", new = "ta", y)

## [1] "Tomorrow I plan to learn about textual analysis."

Note that chartr() replaces every identified letter for replacement so the only time I use it is when I am certain that I want to change every possible occurence of a letter.

### String Abbreviations

To abbreviate strings you can use abbreviate():

streets <- c("Main", "Elm", "Riverbend", "Mario", "Frederick")  
  
# default abbreviations  
abbreviate(streets)

## Main Elm Riverbend Mario Frederick   
## "Main" "Elm" "Rvrb" "Mari" "Frdr"

# set minimum length of abbreviation  
abbreviate(streets, minlength = 2)

## Main Elm Riverbend Mario Frederick   
## "Mn" "El" "Rv" "Mr" "Fr"

Note that if you are working with U.S. states, R already has a pre-built vector with state names (state.name). Also, there is a pre-built vector of abbreviated state names (state.abb).

### Extract/Replace Substrings

To extract or replace substrings in a character vector there are three primary base R functions to use: substr(), substring(), and strsplit(). The purpose of substr() is to extract and replace substrings with specified starting and stopping characters:

alphabet <- paste(LETTERS, collapse = "")  
  
# extract 18th character in string  
substr(alphabet, start = 18, stop = 18)

## [1] "R"

# extract 18-24th characters in string  
substr(alphabet, start = 18, stop = 24)

## [1] "RSTUVWX"

# replace 1st-17th characters with `R`  
substr(alphabet, start = 19, stop = 24) <- "RRRRRR"  
alphabet

## [1] "ABCDEFGHIJKLMNOPQRRRRRRRYZ"

The purpose of substring() is to extract and replace substrings with only a specified starting point. substring() also allows you to extract/replace in a recursive fashion:

alphabet <- paste(LETTERS, collapse = "")  
  
# extract 18th through last character  
substring(alphabet, first = 18)

## [1] "RSTUVWXYZ"

# recursive extraction; specify start position only  
substring(alphabet, first = 18:24)

## [1] "RSTUVWXYZ" "STUVWXYZ" "TUVWXYZ" "UVWXYZ" "VWXYZ" "WXYZ"   
## [7] "XYZ"

# recursive extraction; specify start and stop positions  
substring(alphabet, first = 1:5, last = 3:7)

## [1] "ABC" "BCD" "CDE" "DEF" "EFG"

To split the elements of a character string use strsplit():

z <- "The day after I will take a break and drink a beer."  
strsplit(z, split = " ")

## [[1]]  
## [1] "The" "day" "after" "I" "will" "take" "a" "break"  
## [9] "and" "drink" "a" "beer."

a <- "Alabama-Alaska-Arizona-Arkansas-California"  
strsplit(a, split = "-")

## [[1]]  
## [1] "Alabama" "Alaska" "Arizona" "Arkansas" "California"

Note that the output of strsplit() is a list. To convert the output to a simple atomic vector simply wrap in unlist():

unlist(strsplit(a, split = "-"))

## [1] "Alabama" "Alaska" "Arizona" "Arkansas" "California"

## String manipulation with stringr

The [stringr](http://cran.r-project.org/web/packages/stringr/index.html) package was developed by Hadley Wickham to act as simple wrappers that make R's string functions more consistent, simple, and easier to use. To replicate the functions in this section you will need to install and load the stringr package:

# install stringr package  
install.packages("stringr")  
  
# load package  
library(stringr)

### Basic Operations

There are three string functions that are closely related to their base R equivalents, but with a few enhancements:

* Concatenate with [str\_c()](#str_c)
* Number of characters with [str\_length()](#str_length)
* Substring with [str\_sub()](#str_sub)

{#str\_c} str\_c() is equivalent to the paste() functions:

# same as paste0()  
str\_c("Learning", "to", "use", "the", "stringr", "package")

## [1] "Learningtousethestringrpackage"

# same as paste()  
str\_c("Learning", "to", "use", "the", "stringr", "package", sep = " ")

## [1] "Learning to use the stringr package"

# allows recycling   
str\_c(letters, " is for", "...")

## [1] "a is for..." "b is for..." "c is for..." "d is for..." "e is for..."  
## [6] "f is for..." "g is for..." "h is for..." "i is for..." "j is for..."  
## [11] "k is for..." "l is for..." "m is for..." "n is for..." "o is for..."  
## [16] "p is for..." "q is for..." "r is for..." "s is for..." "t is for..."  
## [21] "u is for..." "v is for..." "w is for..." "x is for..." "y is for..."  
## [26] "z is for..."

{#str\_length} str\_length() is similiar to the nchar() function; however, str\_length() behaves more appropriately with missing ('NA') values:

# some text with NA  
text = c("Learning", "to", NA, "use", "the", NA, "stringr", "package")  
  
# compare `str\_length()` with `nchar()`  
nchar(text)

## [1] 8 2 NA 3 3 NA 7 7

str\_length(text)

## [1] 8 2 NA 3 3 NA 7 7

{#str\_sub} str\_sub() is similar to substr(); however, it returns a zero length vector if any of its inputs are zero length, and otherwise expands each argument to match the longest. It also accepts negative positions, which are calculated from the left of the last character.

x <- "Learning to use the stringr package"  
  
# alternative indexing  
str\_sub(x, start = 1, end = 15)

## [1] "Learning to use"

str\_sub(x, end = 15)

## [1] "Learning to use"

str\_sub(x, start = 17)

## [1] "the stringr package"

str\_sub(x, start = c(1, 17), end = c(15, 35))

## [1] "Learning to use" "the stringr package"

# using negative indices for start/end points from end of string  
str\_sub(x, start = -1)

## [1] "e"

str\_sub(x, start = -19)

## [1] "the stringr package"

str\_sub(x, end = -21)

## [1] "Learning to use"

# Replacement  
str\_sub(x, end = 15) <- "I know how to use"  
x

## [1] "I know how to use the stringr package"

### Duplicate Characters within a String

A new functionality that stringr provides in which base R does not have a specific function for is character duplication:

str\_dup("beer", times = 3)

## [1] "beerbeerbeer"

str\_dup("beer", times = 1:3)

## [1] "beer" "beerbeer" "beerbeerbeer"

# use with a vector of strings  
states\_i\_luv <- state.name[c(6, 23, 34, 35)]  
str\_dup(states\_i\_luv, times = 2)

## [1] "ColoradoColorado" "MinnesotaMinnesota"   
## [3] "North DakotaNorth Dakota" "OhioOhio"

### Remove Leading and Trailing Whitespace

A common task of string processing is that of parsing text into individual words. Often, this results in words having blank spaces (whitespaces) on either end of the word. The str\_trim() can be used to remove these spaces:

text <- c("Text ", " with", " whitespace ", " on", "both ", " sides ")  
  
# remove whitespaces on the left side  
str\_trim(text, side = "left")

## [1] "Text " "with" "whitespace " "on" "both "   
## [6] "sides "

# remove whitespaces on the right side  
str\_trim(text, side = "right")

## [1] "Text" " with" " whitespace" " on" "both"   
## [6] " sides"

# remove whitespaces on both sides  
str\_trim(text, side = "both")

## [1] "Text" "with" "whitespace" "on" "both"   
## [6] "sides"

### Pad a String with Whitespace

To add whitespace, or to *pad* a string, use str\_pad(). You can also use str\_pad() to pad a string with specified characters.

str\_pad("beer", width = 10, side = "left")

## [1] " beer"

str\_pad("beer", width = 10, side = "both")

## [1] " beer "

str\_pad("beer", width = 10, side = "right", pad = "!")

## [1] "beer!!!!!!"

## Set operatons for character strings

There are also base R functions that allows for assessing the set union, intersection, difference, equality, and membership of two vectors.

### Set Union

To obtain the elements of the union between two character vectors use union():

set\_1 <- c("lagunitas", "bells", "dogfish", "summit", "odell")  
set\_2 <- c("sierra", "bells", "harpoon", "lagunitas", "founders")  
  
union(set\_1, set\_2)

## [1] "lagunitas" "bells" "dogfish" "summit" "odell" "sierra"   
## [7] "harpoon" "founders"

### Set Intersection

To obtain the common elements of two character vectors use intersect():

intersect(set\_1, set\_2)

## [1] "lagunitas" "bells"

### Identifying Different Elements

To obtain the non-common elements, or the difference, of two character vectors use setdiff():

# returns elements in set\_1 not in set\_2  
setdiff(set\_1, set\_2)

## [1] "dogfish" "summit" "odell"

# returns elements in set\_2 not in set\_1  
setdiff(set\_2, set\_1)

## [1] "sierra" "harpoon" "founders"

### Testing for Element Equality

To test if two vectors contain the same elements regardless of order use setequal():

set\_3 <- c("woody", "buzz", "rex")  
set\_4 <- c("woody", "andy", "buzz")  
set\_5 <- c("andy", "buzz", "woody")  
  
setequal(set\_3, set\_4)

## [1] FALSE

setequal(set\_4, set\_5)

## [1] TRUE

### Testing for *Exact* Equality

To test if two character vectors are equal in content and order use identical():

set\_6 <- c("woody", "andy", "buzz")  
set\_7 <- c("andy", "buzz", "woody")  
set\_8 <- c("woody", "andy", "buzz")  
  
identical(set\_6, set\_7)

## [1] FALSE

identical(set\_6, set\_8)

## [1] TRUE

### Identifying if Elements are Contained in a String

To test if an element is contained within a character vector use is.element() or %in%:

good <- "andy"  
bad <- "sid"  
  
is.element(good, set\_8)

## [1] TRUE

good %in% set\_8

## [1] TRUE

bad %in% set\_8

## [1] FALSE

### Sorting a String

To sort a character vector use sort():

sort(set\_8)

## [1] "andy" "buzz" "woody"

sort(set\_8, decreasing = TRUE)

## [1] "woody" "buzz" "andy"